Unix File System Calls

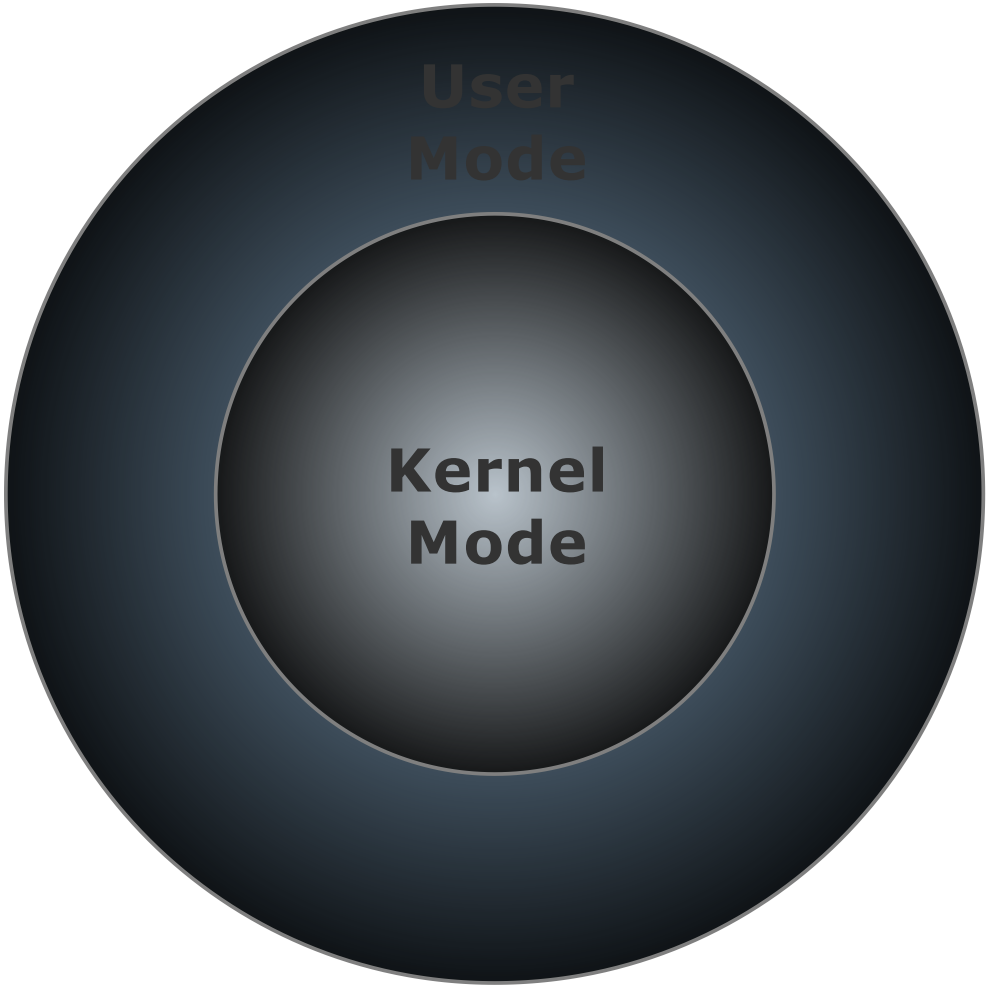
BITS Pilani
Pilani Campus

Department of Computer Science and Information Systems

Points to be Covered

- User Process
- Kernel Process
- System Calls
- Hierarchy of System Calls
  - Creat
  - Open
  - Close
  - Read
  - Write
  - Lseek

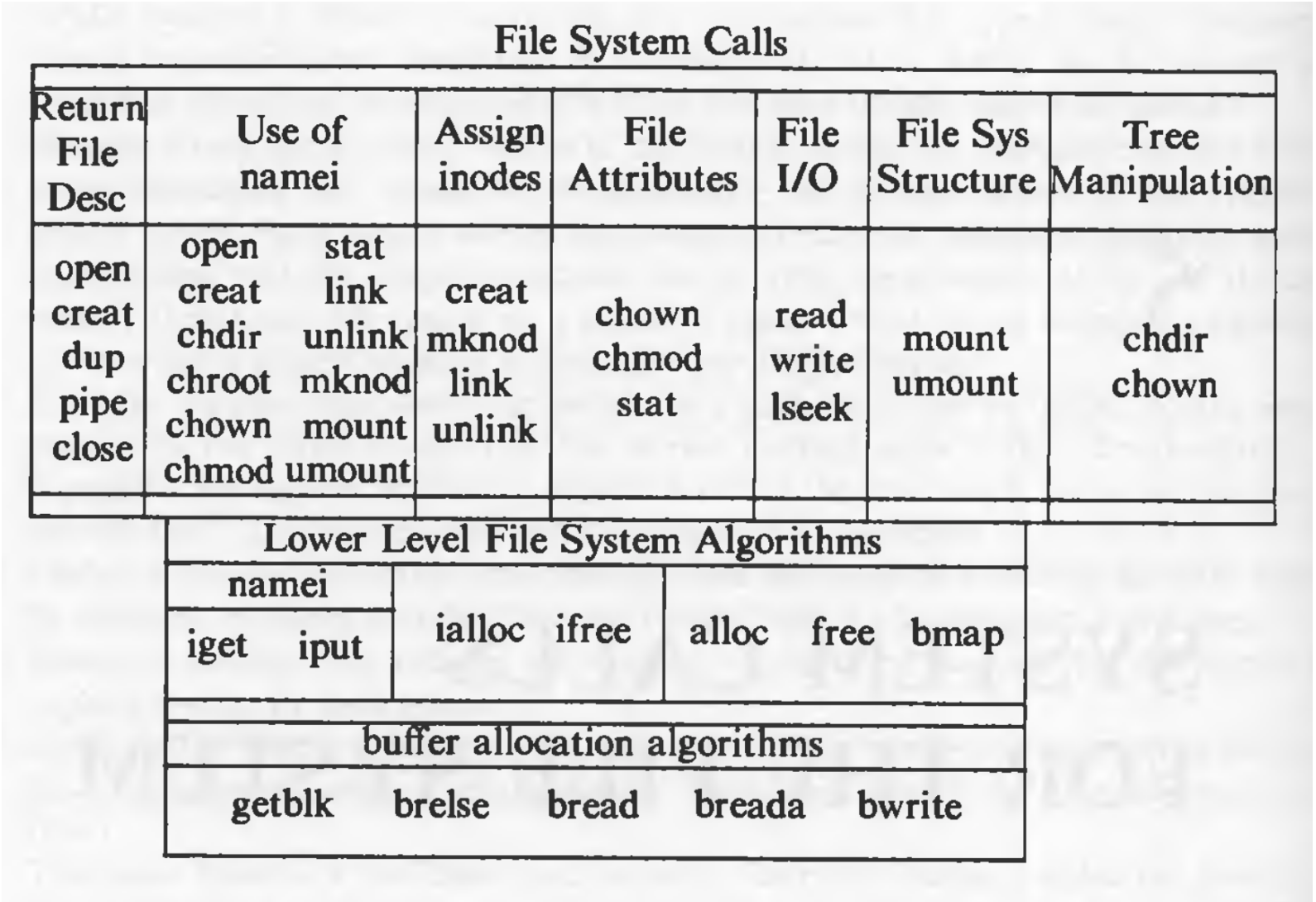
Modes of Operating System



File System Calls

System Calls	It is a programmatic way in which a computer program requests a service from the kernel of the operating system.
	It provides an interface between a process and operating system to allow user-level processes to request services of the operating system.
	System calls are the only entry points into the kernel system.

File System Calls



File Descriptor

- **File descriptor: FD** is integer that uniquely identifies an open file of the process.

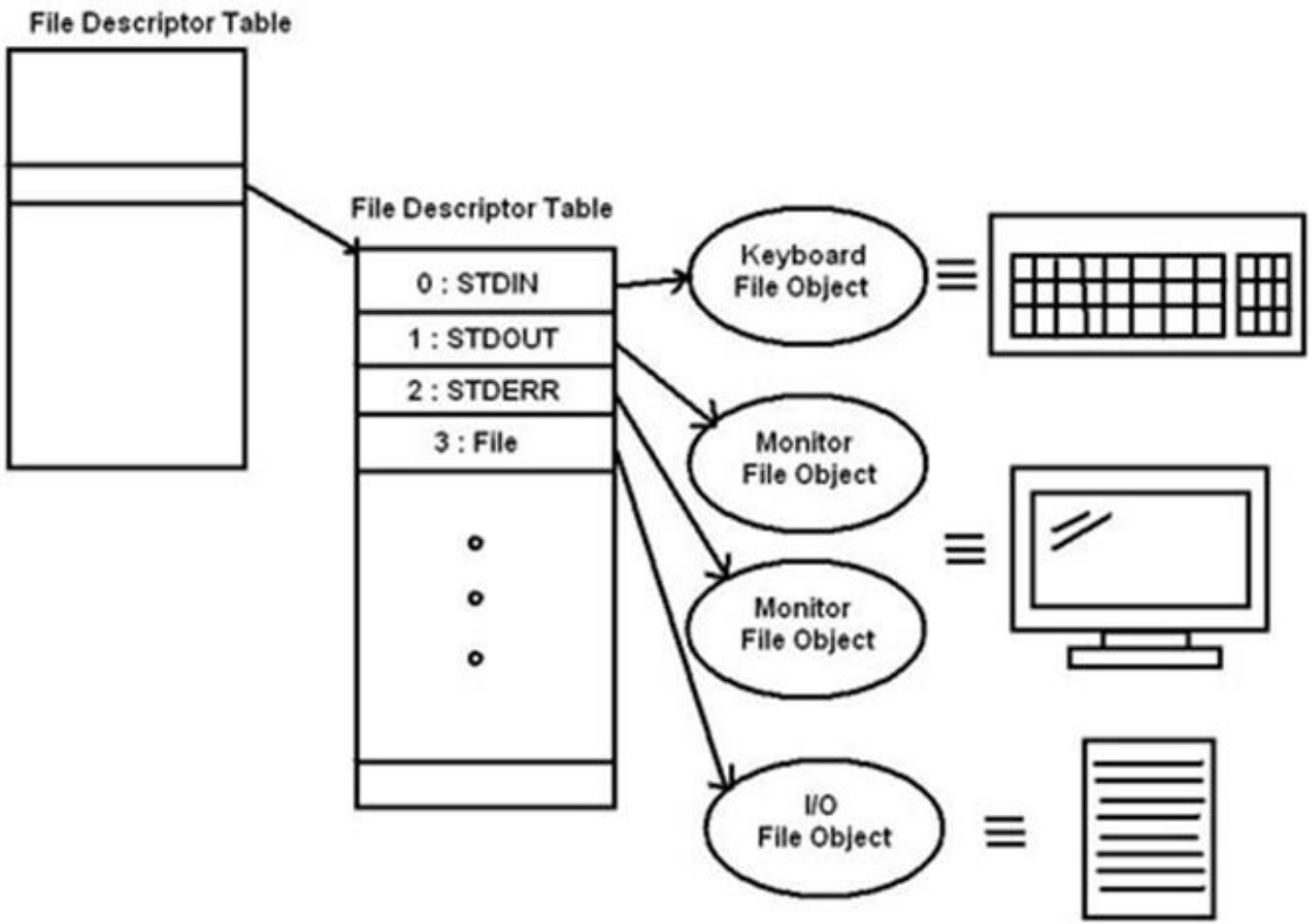
- **File Descriptor table**: **FDT** is the collection of integer array indices that are file descriptors in which elements are pointers to file table entries. One unique file descriptors table is provided in operating system for each process.

- **File Table Entry: FDE** is a structure In-memory surrogate for an open file, which is created when process request to opens file and these entries maintains file position.

File Descriptor Table

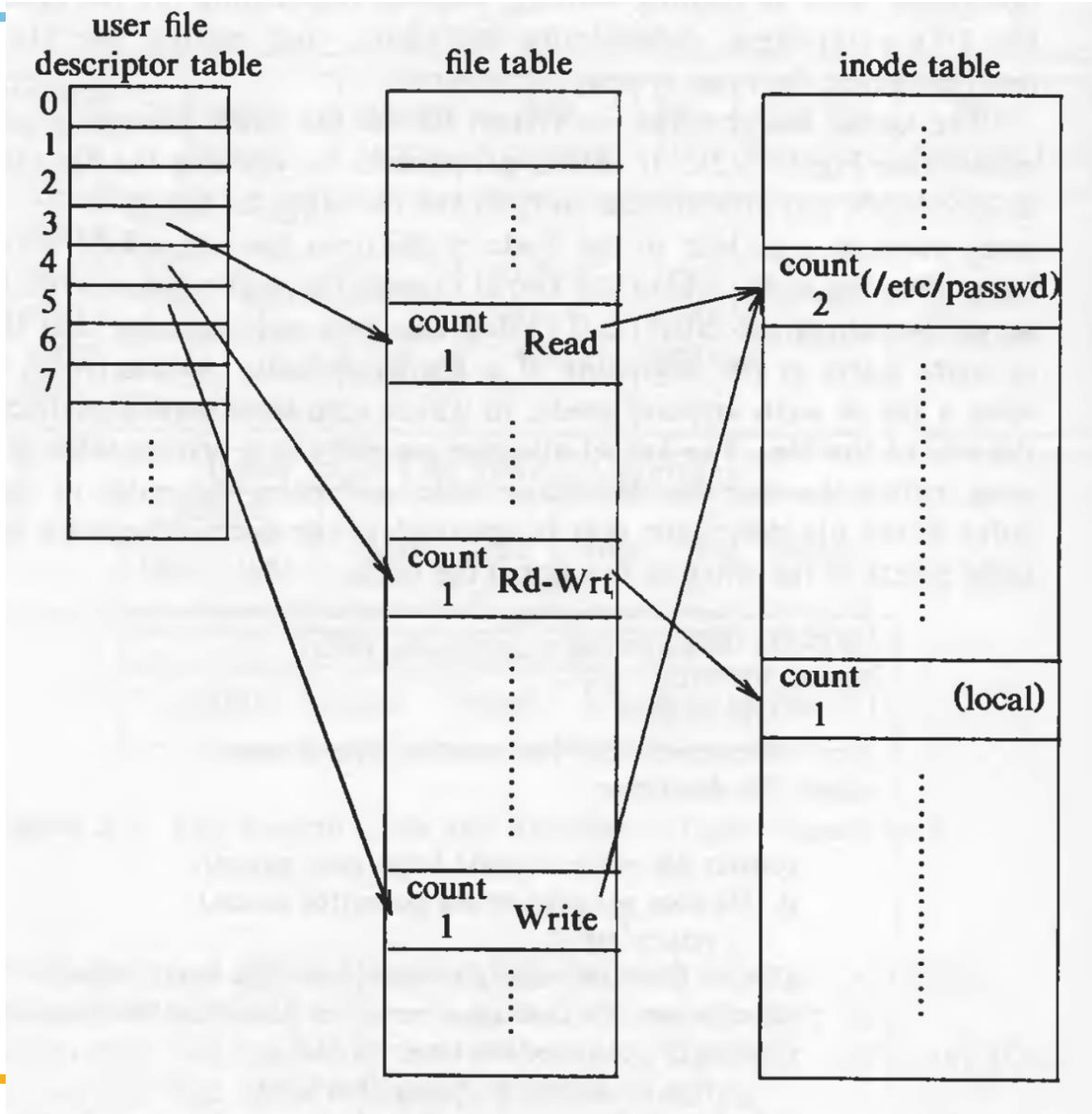
- **the fd table is an array**
  - for one running process
  - associated with every process
  - list of open files stored in the fd table
  - each entry corresponds to one open file
  - that is copied when fork is executed
- **e.g.:**
  - each process has its own table
  - open: add an entry to the table (new file created)
  - close: delete an entry from the table
  - this is for a "normal" process, one with no redirection of i/o (no forks)
  - fd # from 0 to 31

File Descriptor Table



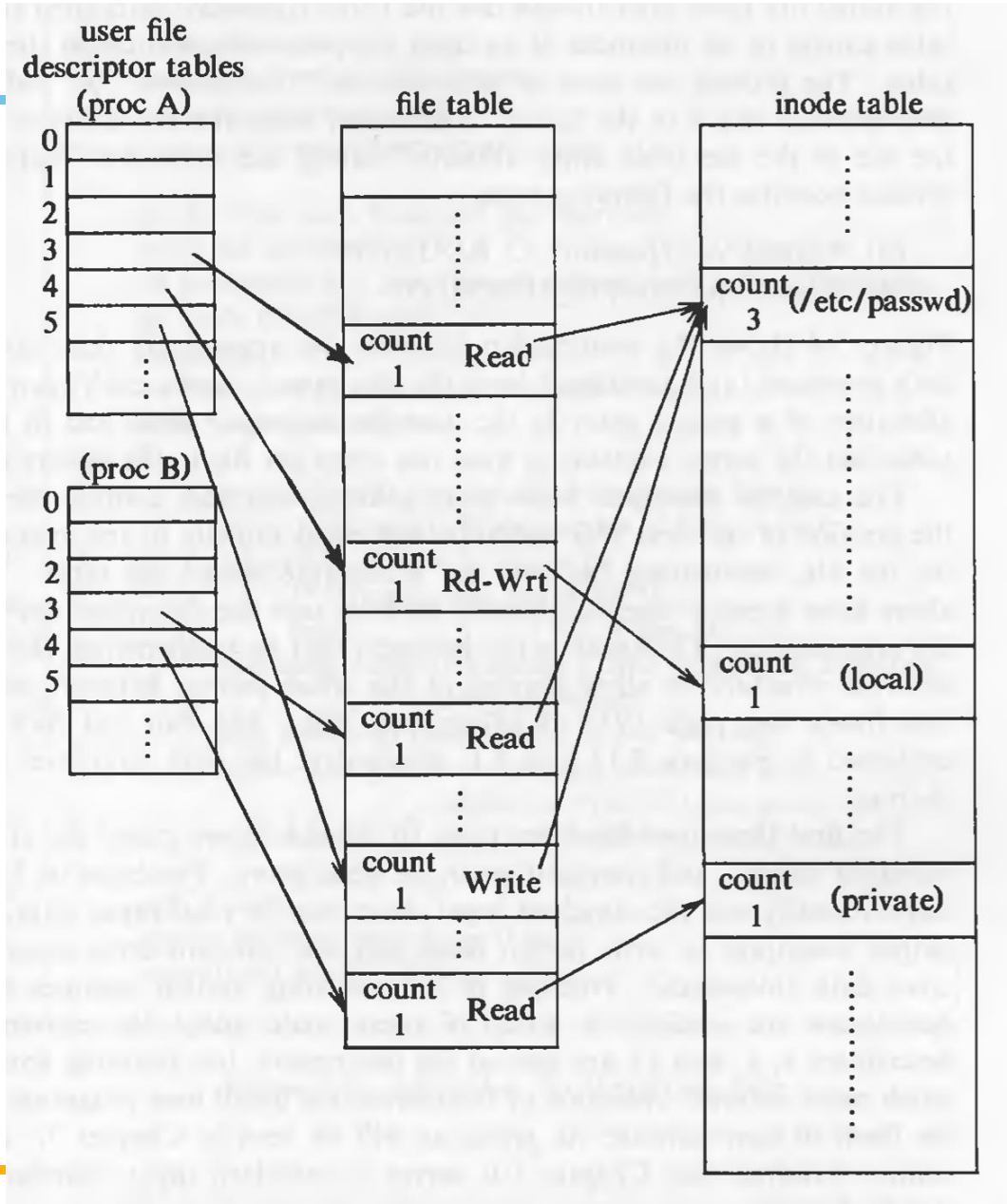
If a process executes following code:

```
fd1 = open ("/etc/passwd", O_RDONLY);
fd2 = open ("local", O_RDWR);
fd3 = open ("/etc/passwd", O_WRONLY);
```



If another process (say process B) executes the following code:

```
fd1 = open ("/etc/passwd", O_RDONLY);
fd2 = open ("private", O_RDONLY);
```



Lower level File System Algorithms

Name of Algorithm	Operation
namei	Parses the path name one component at a time and returns the inode of the input path name.
iget	Allocates the in-core copy of the inode if it exist and locks it. The inode is returned with reference count 1 greater than previous.
iput	Releases the inode by decrementing the reference count. Unlocks the inode to provide access to other system calls. Stores back if in-core copy is different from disk copy.
bmap	Converts a file byte offset into a physical disk block.
ialloc	Allocates the inode for a new file from the free list of inodes
ifree	If reference count becomes 0, the inode is released and added to the free list of inodes
alloc	Allocates the disk inode
free	Releases the disk inode

File System Calls

System calls	Function
creat	Create a new empty file
open	open an existing file or create a new file
read	Read data from a file
write	Write data to a file
lseek	Move the read/write pointer to the specified location
close	Close an open file
unlink	Delete a file
chmod	Change the file protection attributes
stat	Read file information from inodes

creat() system call

Syntax in C language: int creat(char *filename, mode_t mode)

Parameter :

- **filename :** name of the file which you want to create
- **mode :** indicates permissions of new file.

Returns :

- return first unused file descriptor (generally 3 when first creat use in process because 0, 1, 2 fd are reserved)
- return -1 when error

How it work in OS

- Create new empty file on disk
- Create file table entry
- Set first unused file descriptor to point to file table entry
- Return file descriptor used, -1 upon failure

File Permissions

There are three Classes – Owner, Group, Others.

- The **Owner** is the usually the creator of the files/folders. In Linux, files or folders that you created in your Home directory are usually *owned* by you, unless you specifically change the ownership.
- The **Group** contains a group of users who share the same permissions and user privilege.
- **Others** means the general public.

0 – no permission
1 – execute
2 – write
3 – write and execute
4 – read
5 – read and execute
6 – read and write
7 – read, write, and execute

File Permissions

What about the 3 digits '777'?

Well, the first digit is assigned to the Owner, the second digit is assigned to the Group and the third digit is assigned to the Others. So for a file with '777' permission, everyone can read, write and execute the file. Here are some of the commonly used permissions:

- **755** – This set of permission is commonly used in web server. The owner has all the permissions to read, write and execute. Everyone else can only read and execute, but cannot make changes to the file.
- **644** – Only the owner can read and write. Everyone else can only read. No one can execute the file.
- **655** – Only the owner can read and write, but not execute the file. Everyone else can read and execute, but cannot modify the file.

File Permissions

Symbolic Notation	Numeric Notation	English
----------	0000	no permissions
-rwx------	0700	read, write, & execute only for owner
-rwxrwx---	0770	read, write, & execute for owner and group
-rwxrwxrwx	0777	read, write, & execute for owner, group and others
---x--x--x	0111	execute
--w--w--w-	0222	write
--wx-wx-wx	0333	write & execute
-r--r--r--	0444	read
-r-xr-xr-x	0555	read & execute
-rw-rw-rw-	0666	read & write
-rwxr-----	0740	owner can read, write, & execute; group can only read; others have no permissions

creat() system call

```
# include <stdio.h>

int main ()
{
        int fd1, fd2;
        printf("/nThis would create two files");
        fd1= creat("txt1.txt", 0777);
        fd2= creat("txt2.txt", 0777);
}
```

open system call

- Syntax is

  fd = open(pathname, flags, modes);

  - int open(const char *path, int flags, mode_t mode);
  - int open(const char *path, int flags);

  where,

  - pathname is the file name
  - flags indicate type of file (reading/writing)
  - modes gives the file permission (if file is created)
  - Returns an integer called file descriptor
  - Rest of the system calls make use of this file descriptor.

Different flags values

Flag	Description
O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_NONBLOCK	do not block on open
O_APPEND	append on each write
O_CREAT	create file if it does not exist
O_TRUNC	truncate size to 0
O_EXCL	error if create and file exists
O_SHLOCK	atomically obtain a shared lock
O_EXLOCK	atomically obtain an exclusive lock
O_DIRECT	eliminate or reduce cache effects
O_FSYNC	synchronous writes
O_NOFOLLOW	do not follow symlinks

Example on open()

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
        int fd1, fd2;
        fd 1 = open("txt1.txt", O_RDONLY | O_CREAT,  0777);
        fd 2 = open("txt2.txt", O_RDONLY | O_CREAT,  0777);
}
```

Example on open()

```
int main(int argc, char *argv[])
{
        int fd1;
         fd1 = open(argv[1],O_RDONLY);
          if(fd1 == -1){
                printf("Error opening file \n");
                exit(0);
            }
          printf("file opened successfully\n");
          printf("fd1=%d\n",fd1);
}
```

```
CSIS@localhost myopen]$ ./exam1  t2.txt
Error opening file
[CSIS@localhost myopen]$ ./exam1  t1.txt
file opened successfully
fd1=3
```

Close ()

- int close(int fd); /*file descriptor */

  /* Returns 0 on success and -1 on error */

- It makes the file descriptor available for re-use.
- It does not flush any kernel buffers or perform any other clean-up task.

Close ()

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

int main()
{
        int fd1, ret;
        fd1 = open("txt1.txt", O_RDONLY | O_CREAT,  0777);
        ret = close(fd1);
        printf("\n Result:%d", ret);
}
```

read()

Syntax:

size_t read(int fd, void *buf, size_t nbytes);

fd – file descriptor

nbytes – number of bytes to be read

buf – buffer to hold data after read

- The function returns the number of bytes read, 0 for end of file (EOF) and -1 in case an error occurred.
- The process that executes a read operation waits until the system puts the data from the disk into the buffer.

read() system call

```
int main (int argc, char *argv[])
{
    char buff[10];
    int ret;

   fd3 = open(“file.txt", O_RDONLY);
    if (fd3 < 0) { perror("r1"); exit(1); }

    ret = read(fd3, buff, 10);
    printf(“%s\n”,buff);

    return 0;
}
```

write()

Syntax:

size_t write(int fd, void *buf, size_t nbytes);

fd – file descriptor

buf – buffer to hold data for write operation

nbytes – number of bytes to be write operation

The function returns the number of bytes written and the value -1 in case of an error.

write system call

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
        char buffer[12] = “I LOVE BITS”
        int fd1, ret;
        fd1 = creat(“txt1.txt”, 0777);
        ret = write(fd1, buffer,  sizeof(buffer));
        return 0;
}
```

lseek() system call

- Move the read/write pointer to the specified location for next read/write operation.

Syntax: off_t lseek(int fd, off_t offset, int reference);

- offset is used to specify the position
- reference is used by the offset
  - SEEK_SET – offset is absolute position
  - SEEK_CUR – offset is relative to the current position
  - SEEK_END – offset is relative to the end of the file

lseek() system call

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
 int main(){
 int file;
 if((file=open("testfile.txt",O_RDONLY)) < 0)  return -1;
 char buffer[19];
 if(read(file,buffer,19) != 19)  return -1;
 printf("%s\n",buffer);
 if(lseek(file,10,SEEK_SET) < 0) return -1;
 if(read(file,buffer,19) != 19)  return -1;
 printf("%s\n",buffer);
 return 0;
}
```

Testfile.txt

This is a text file that will be used to demonstrate the use of lseek.

Output: This is a text file
text file that will

Any Queries?